1

AFIT/EN-TR-91-8

Air Force Institute of Technology

Developing Object-Oriented User Interfaces in Ada with the X Window System

Gary W. Klabunde, Capt, USAF
Mark A. Roth, Maj, USAF

27 December 1991

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

Developing Object-Oriented User Interfaces in Ada With the X Window System

Gary W. Klabunde[†]
Mark A. Roth[‡]

Abstract

The graphical user interface has been accepted as being one of the most important parts of user interactive software projects. Until recently, however, the design and implementation of such interfaces in Ada was a long and difficult process. A solution to this problem was found with the introduction of the X Window System in the mid 1980's. These C libraries make it relatively easy to construct sophisticated user interfaces. In the last couple of years, several organizations have developed bindings to, or implementations of, the X Window System software for use in Ada programs. This paper discusses these efforts in general and concentrates on two sets of bindings that were used in the development of a graphical user interface for a computer wargame.

1 Introduction

The user interface is the component of the application through which the user's actions are translated into one or more requests for services of the applications, and that provides feedback concerning the outcome of the requested actions [9]. Because of the importance of this interaction, the design of efficient and easy to use user interfaces is receiving increased attention. Most people now realize that if an application has a user interface that is "unfriendly" or difficult to use, it is probably going to sit on the shelf unused. Also, user interfaces using some type of windowing system are fast becoming a common feature of most computer systems. As a result, users tend to expect all application programs to have a professional, polished user-friendly interface [16]. Most programming languages, including the Ada language, have only rudimentary input and output (I/O) capabilities. As such, user interface programmers had to develop some other methods for anything except simple line or character I/O.

The introduction of the X Window System in the mid 1980's changed the way user interfaces were developed. The X Window System, or X, is a collection (library) of subroutines, written in the C language, that allows for the creation and manipulation of graphical user interfaces using multiple windows. These subroutines provide the mechanism to achieve the goals previously discussed.

Recognizing the importance of X to the development of user interfaces, some members of the Ada community began working on ways to access the X Window System from within Ada programs. The first efforts involved developing bindings to the X routines. Subsequent efforts have looked at ways to implement X in the Ada language.

This paper briefly discusses the X Window System and some of the more significant accomplishments in accessing X from Ada programs. Particular attention is paid to the bindings developed by Stephen Hyland formerly of Science Applications International Corporation and E.J. Jones of Boeing Aerospace Corporation. We discuss how these bindings were successfully used at the Air Force Institute of Technology for the design and implementation of a user interface for the Saber computer wargame. We describe how

[†]Gary Klabunde's work was done while with the Department of Electrical and Computer Engineering, Air Force Institute of Technology. He is currently with the Strategic Communications and Computer Center (SCCC/SOSW), Offut AFB, NE.

[‡]Mark Roth is with the Department of Electrical and Computer Engineering (AFIT/ENG), Air Force Institute of Technology, Wright-Patterson AFB, OH 45433.

to create new bindings along with a summary of problems encountered when combining various sets of bindings in a single application. The paper ends with a description of the impact of the X Window System on an object-oriented design and some of the limitations of the bindings.

2 The X Window System

The X Window System, or X, is a device independent, network transparent windowing system that allows for the development of portable graphical user interfaces [10, 11, 16]. It was developed in the mid 1980's at the Massachusetts Institute of Technology (MIT) in response to a need to execute graphical software on several different types of incompatible workstations. Robert Scheifler of MIT and James Gettys of Digital Equipment Corporation (DEC) developed X with the primary goals of portability and extensibility [11]. Another major consideration was to restrict the applications developer as little as possible. As a result, X "...provides mechanism rather than policy" [5:xvii].

To achieve these goals, the X Window System relies on the fundamental principles of network transparency and a request/event system. In X, each workstation that is to display graphical information (i.e., windows or their contents) must have a process called the X server. According to Douglas Young, the X server "...creates and manipulates windows on the screen, produces text and graphics, and handles input devices such as a keyboard and mouse" [16:2]. A client, on the other hand, is any application program that uses the services of the X server. The clients and servers communicate with each other by sending requests and event notifications over a network.

When a client wants to perform some action on the display, it communicates this desire by issuing a request to the appropriate X server. Young states:

> Clients typically request the server to create, destroy, or reconfigure windows, or to display text or graphics in a window. Clients can also request information about the current state of windows or other resources. [16:4]

The X server, conversely, communicates with clients by issuing event notifications. Event notifications are sent in response to such user actions as moving a mouse into a window, by pressing a mouse button, or pressing a key on the keyboard. The X server also sends event notifications when the state of a window changes [16]. Applications programs act on these events by registering callbacks with the X Window System. A callback is simply a procedure or function that is to be executed when a specific event occurs.

2.1 Xlib

The X Window System was designed to provide the mechanisms for the application program to control what is seen on the display screen. The programmer is not constrained by any particular user interface policy. These mechanisms are embodied in a library of C functions known as Xlib. The Xlib routines allow for client control over the display, windows, and input devices. Additionally, the functions provide the capability for clients to design such things as menus, scroll bars, and dialog boxes.

2.2 Toolkits

While applications programmers can use the Xlib routines to accomplish any task in X, many find the low-level routines tedious and difficult to use. Jay Tevis[12] noted that the simple action of creating and customizing a new window on the display takes at least 24 calls to Xlib. To simplify the development of applications programs, many toolkits have been developed. Toolkits can be viewed as libraries of graphical programs layered on top of Xlib. They were designed to hide the details of Xlib, making it easier to develop X applications.

There are several toolkits available today. Some of the better known ones include: the X Toolkit (Xt) from MIT, the Xrlib Toolkit (Xr) from Hewlett-Packard (HP), Open Look and XView from Sun

Microsystems, and Andrew from Carnegie Mellon University. Of those listed, Xt is one of the most popular [4]. Along with Xlib, it is delivered as a standard part of the X Window System.

Xt is an object-oriented toolkit used to build the higher level components that make up the user interface [4]. It consists of a layer called the Xt Intrinsics along with a collection of user interface components called widgets. Widget sets typically consist of objects such as scroll bars, title bars, menus, dialog boxes and buttons. In keeping with the X philosophy, the Xt Intrinsics layer remains policy free. As such, it only provides mechanisms that do not affect the "look and feel" (outward appearance and behavior) of the user interface [16]. These mechanisms allow for the creation and management of reusable widgets. It is this extensibility along with its object-oriented design that makes the X Toolkit attractive to user interface designers [14].

It is the programmer's choice of a widget set that determines the high-level "look and feel" of the user interface. Just as there is no "standard" toolkit, there are many different widget sets supported by Xt Intrinsics. However, as Young writes, "...from an application programmer's viewpoint, most widget sets provide similar capabilities" [16:12]. Some of the more popular widget sets include the Athena widget set from MIT, the X Widget set from HP, and the Motif widgets from the Open Software Foundation.

The Open Software Foundation (OSF) was formed in 1988 by a group of UNIX vendors including, among others, IBM, HP, and Sun Microsystems. The Motif widget set they created is designed to run on such platforms as DEC, HP, IBM, Sun, and Intel 80386 based architectures [6]. Eric Johnson lists three advantages to using Motif [6:4]:

1. Motif provides a standard interface with a consistent look and feel. Your users will have less work to do in learning other Motif applications, since much of the work learning other Motif applications will translate directly to your applications.
2. Motif provides a very high-level object-oriented library. You can generate extremely complex graphical programs with a very small amount of code.
3. Motif has been adopted by many of the major players in the computer industry. Many of your customers are probably using Motif right now. You'll do a better job selling to them if your applications are also based on Motif.

Structurally, the Xt Intrinsics is built on top of Xlib. The Motif widget set, in turn, relies on the functions provided by the Xt Intrinsics. A typical application program may make calls to the widget set, the Xt Intrinsics, or Xlib itself during its execution. This configuration is illustrated in Figure 1.

Many user interface designers elect to design their own widget sets. Some do it for the challenge, while others design their own widgets out of necessity. A user interface designer may have a need for a special widget not provided by any available widget sets. However, designing custom widgets decreases the portability of the user interface code and of the application code in general [4].

3 Ada and the X Window System

Originally, Xlib, Xt Intrinsics and most widget sets were written in the programming language C. Until a few years ago, there was no way for an application program written in Ada to use the X Window System. Recent efforts have taken two approaches: Ada bindings to X and Ada implementations of the X libraries. Most of the Ada bindings are tied to particular operating systems and will only work with a particular Ada compiler. The Alsys, Meridian, and Verdix compilers, along with their derivatives, are used most often for the bindings [1].

3.1 Ada Bindings to X

In 1987, the Science Applications International Corporation (SAIC) developed Ada bindings to the Xlib C routines. Their work was performed under a Software Technology for Adaptable Reliable Systems (STARS) Foundation contract, and is therefore in the public domain. According to Kurt Wallnau, "...a substantial

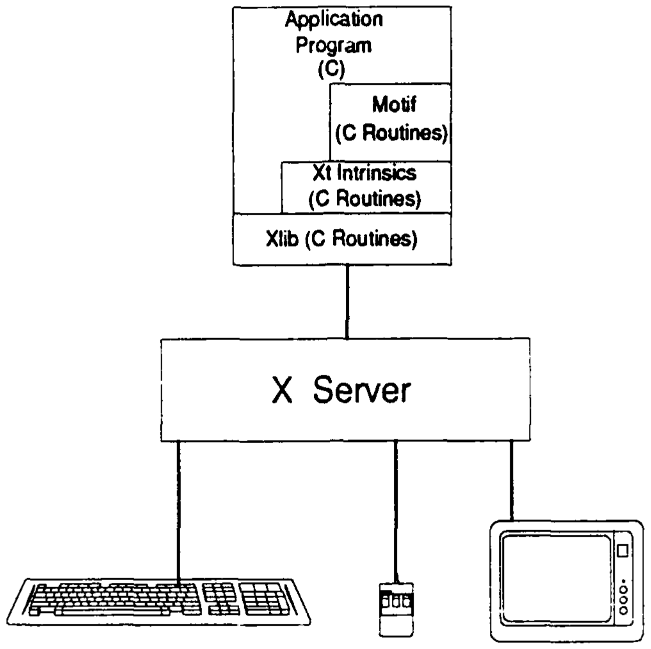

Figure 1: Typical X Windows Configuration

effort was made to map the C data types to Ada, and do as much Xlib processing in Ada as possible before sending the actual request to the C implementation" [14:5]. The actual Ada interface is accomplished through the use of Ada *pragma interface* statements [3]. Put simply, the *pragma interface* construct allows an Ada program to call subprograms written in another language [2]. Figure 2 shows the configuration of an Ada program using the SAIC bindings to interface with Xlib. In this figure, the application program has no access to any toolkits or widget sets.

In a manner similar to that used by SAI the Boeing Corporation recently developed Ada bindings to a large subset of the Xt Intrinsics and the Motif widget set. Their code also provides access to a very limited subset of Xlib functions and data types. Like the SAIC code, Boeing's effort was sponsored by a STARS contract [7]. For the most part, the subroutine names and parameter lists closely mirror the actual C routines. Also, Boeing added a few subprograms to assist in the building of some commonly used parameter lists. The bindings require the Verdix Ada Development System (VADS) version 5.5 or higher to execute. While the documentation on the software is relatively sparse, it does indicate which modules would require changes in order to port the bindings to other systems.

Figure 3 shows the configuration of an Ada program using only the Boeing bindings. The dashed lines indicate that a small portion of the Xt Intrinsics and Motif functions are unavailable to the Ada program. Also, the application program cannot access the majority of the Xlib functions.

The Ada application program accesses the Xt Intrinsics and Motif routines by calling the appropriate subprogram in the bindings. For the most part, the bodies of the called subprograms contain code to convert the Boeing data structures and types to the types needed by the corresponding C code. The subprogram bodies then call internal procedures or functions that are bound to the Xt Intrinsics or Motif routines passing in the converted parameters.

The bindings developed by Boeing and the SAIC are available at no cost to the Department of Defense. Recently, several other corporations have also developed bindings that are available for purchase [1] These companies have basically taken one of two approaches. Some have followed the approach taken by the SAIC and Boeing. Others, such as Hewlett-Packard, took an alternative approach. To alleviate the need for much of the type conversion used by the SAIC and Boeing bindings, Hewlett-Packard binds the Ada

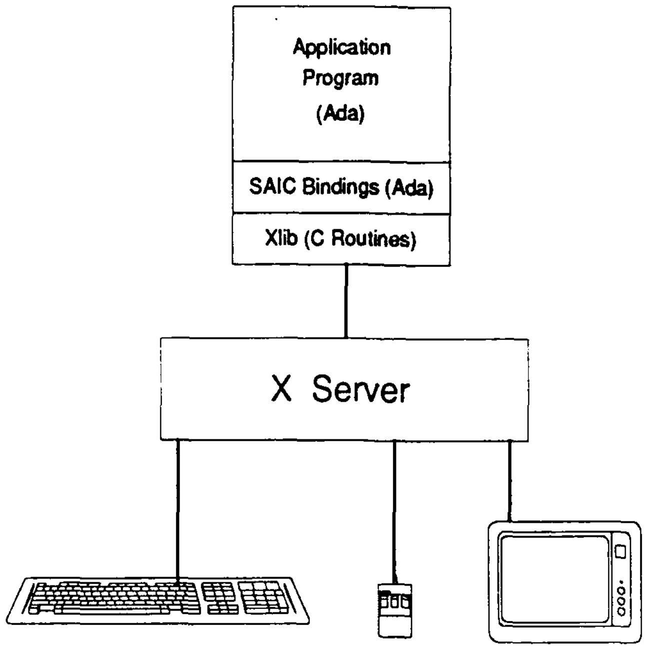

Figure 2: Application Program Configuration Using the SAIC Bindings

subroutines directly to the corresponding C code. This results in very little code in the package bodies. To accomplish this, they make heavy use of Ada *access types*.

3.2 Ada Implementations

The USAF Electronic Systems Division recognized the need to write X Windows application programs in Ada at a higher level than through Xlib alone. In 1989, they sponsored a STARS Foundation contract to further research the capabilities of interfacing Ada and the X Window System [4]. The resulting reports documented efforts at integrating Ada with the X Toolkit (Xt).

As part of this STARS contract, Unisys Corporation developed an Ada implementation of (not bindings to) the X11R3 version of the Xt Intrinsics. "Ada/Xt," as it is called, "provides an intrinsics package which provides the functionality of Xt used to manage X resources, events and hierarchical widget construction" [15:1]. This software package uses a modified and corrected version of the SAIC bindings to interface to Xlib. Ada/Xt also includes a sample widget set consisting of ten Athena widgets and two HP widgets [15].

Unisys elected to develop an Xt implementation rather than Ada bindings, as SAIC did. The reasons for this included [14:9-10]:

1. The issue of widget extensibility. Ada bindings would require that new widgets be programmed in C.
2. The issues of inter-language runtime cooperation.
3. The issues of runtime environment interaction.

Figure 4 represents a typical Ada application program using the Ada/Xt interface. The Ada application code can make use of the provided widgets, make calls to Ada/Xt, or make calls directly to the Xlib via the modified SAIC bindings. Thus, the full flexibility of an X application program written in C is maintained.

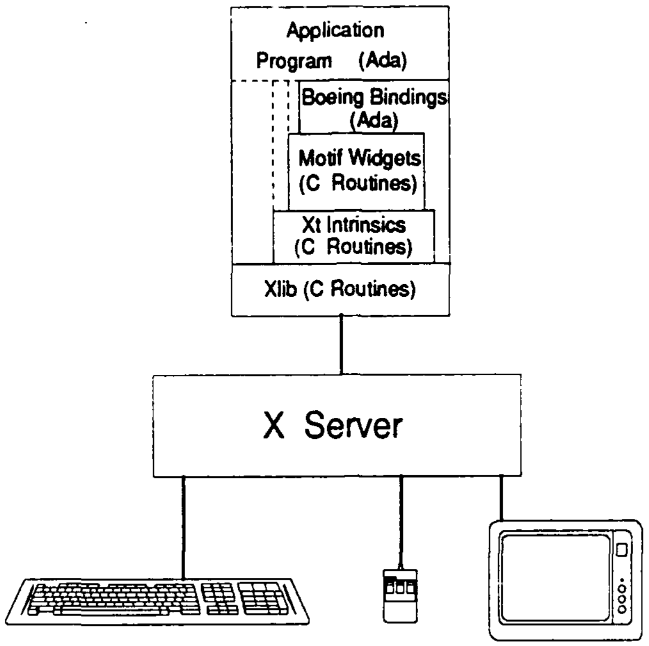
Application Program (Ada)
Boeing Bindings (Ada)
Motif Widgets (C Routines)
Xt Intrinsics (C Routines)
Xlib (C Routines)
X Server

Figure 3: Application Program Configuration Using Boeing's Bindings

4 Saber: A Sample Application

Some user interfaces can be implemented by simply calling subroutines in the Xt Intrinsics and Motif widget set. Others may require additional calls to selected Xlib routines. The object-oriented graphical user interface for the Saber wargame [8] developed at the Air Force Institute of Technology fits into the latter category. Displaying the graphical symbols for the airbases, aircraft missions, and land units required the use of low-level Xlib subroutines.

Due to the need to access the Xlib, Xt Intrinsics and Motif libraries, it was clear that, as a minimum, the SAIC bindings would have to be used. The choice remained of whether to supplement it with the Boeing bindings or the Ada/Xt software developed by Unisys. Using the Ada/Xt software would have required the full or partial development of an Ada implementation of the Motif widget set. The Boeing software, on the other hand, already had bindings developed for Motif. Thus, we decided to utilize the Boeing bindings in combination with the SAIC software to develop the Saber user interface.

The Saber user interface was also designed to use a hexagon (hex) widget designed by the Air Force Wargaming Center (AFWC). This object-oriented widget contains routines to create and manipulate hexboards. Routines are provided to display certain features inside of a hex. These features include rivers, roads, cities, city names, forestation, and background color.

4.1 New Bindings for the Hex Widget

Since the hex widget is written in the C programming language, Ada bindings had to be developed. These hex bindings were modeled after Boeing's bindings to the Motif widget set. Each procedure exported by the hex widget had to have a corresponding Ada procedure. To aid in understanding, the Ada procedure names were given the same names as their C counterparts except that underscores were inserted between words. Thus, the C procedure "Hx_SetHexLabel" became "Hx_Set_Hex_Label". The complete binding for this procedure is shown in Figure 5.

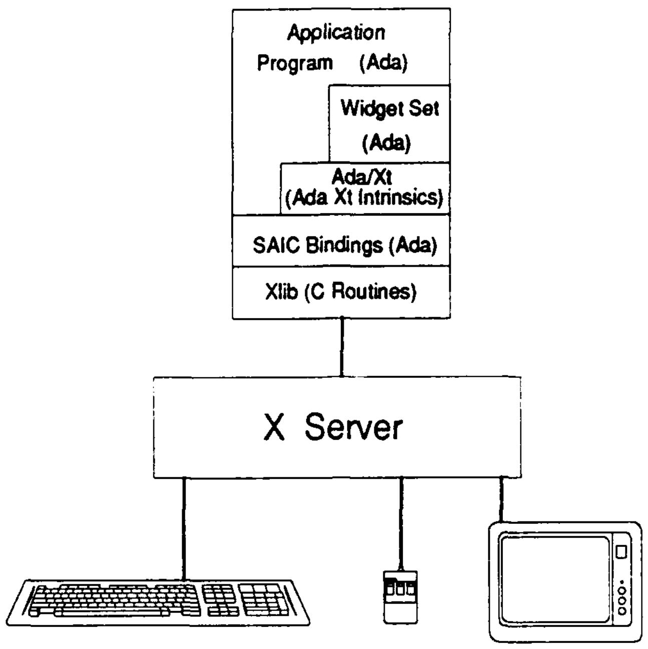

Figure 4: Application Program Configuration Using Unisys' Ada/Xt

As can be seen from the figure, the Ada procedure was implemented with another procedure nested inside of it. The outer procedure is the one called by the application program. Thus, the application program should declare variables of the appropriate type to pass into the procedure. The inner procedure is what is actually bound to the corresponding C procedure. In order to distinguish it to the compiler, it is given the same name as the outer procedure except that all underscores are removed. It should be noted that the inner procedure has no body in the Ada code. Its body is actually the C procedure.

The actual binding was accomplished using the Ada *pragma interface* and *pragma interface_name* constructs. In Figure 5, the *pragma interface* construct indicates that the inner procedure is to be bound to a procedure written in the language C. The name of the Ada procedure is then paired with the name of the corresponding C procedure through the *pragma interface_name* construct.

The primary purpose of the body of the outer procedure is to convert the Ada input parameters to the types needed by the inner procedure for transfer to the C subroutine. However, the challenge in developing the bindings was determining exactly what types of parameters should be passed to the C procedures. Table 1 was developed to assist in this determination for some of the major data types. Given the type and mode of the parameter in the outer procedure, the table lists the type for the variable in the inner procedure. It also shows how the type conversion should be accomplished in the body of the outer procedure.

In general, if a variable in the outer procedure has a mode of "out", then the corresponding variable in the inner procedure must be of type "System.Address". This is because the C procedure must have the address of the variable if it is going to set or change the value. One other important point is illustrated in Figure 5. In C, all strings must be terminated by an ASCII null character. Ada strings, however, typically do not end with this character. Thus, before sending the string address to the C subroutine, the Ada bindings append an ASCII null.

```
procedure HX_Set_Hex_Label( Hex_Widget : in WIDGET;
                            Hex_X      : in AFS_LARGE_NATURAL;
                            Hex_Y      : in AFS_LARGE_NATURAL;
                            Label      : in STRING;
                            Redraw     : in BOOLEAN) is

   procedure HXSetHexLabel( Hex_Widget : in SYSTEM.ADDRESS;
                            Hex_X      : in AFS_LARGE_NATURAL;
                            Hex_Y      : in AFS_LARGE_NATURAL;
                            Label      : in SYSTEM.ADDRESS;
                            Redraw     : in AFS_LARGE_NATURAL );

   pragma INTERFACE (C, HXSetHexLabel);
   pragma INTERFACE_NAME (HXSetHexLabel, "_HX_SetHexLabel");

   Temp_Label : constant STRING := Label & ASCII.NUL;

begin

   HXSetHexLabel( Widget_To_Addr( Hex_Widget ),
                  Hex_X,
                  Hex_Y,
                  Temp_Label(1)'address,
                  BOOLEAN'pos( Redraw ) );

end HX_Set_Hex_Label;
```

Figure 5: Ada Binding to Hx_SetHexLabel

4.2 Combining the Ada Bindings

The relationship between the Saber user interface and the various Ada bindings is shown in Figure 6. This figure accurately reflects that the Boeing software contains bindings to a small subset of the Xlib functions in addition to the bindings to the Xt Intrinsics and Motif widget set. The user interface may make calls to the Boeing bindings, the SAIC bindings, and the hex widget bindings. In fact, interactions between the application program and the X Window System are made solely through these bindings.

The Boeing bindings were the primary means of interfacing with the X Window System, while the SAIC bindings were used primarily for the creation of the graphical unit symbols. Making the few calls to the SAIC bindings was not straightforward because of inconsistent types used by the two sets of bindings. Some inconsistencies were resolved by simple type conversion while others required the addition of new subroutines to the software.

4.2.1 Type Conversions.

By necessity, the Boeing software contains Ada declarations of a few low-level Xlib routines. These declarations for such things as the X Window System display, windows, and drawables were needed because the Xt Intrinsics provides functions to return these values that are created when the connection with the X server is established and windows are displayed on the screen.

Several of the SAIC procedures used to create the unit symbol pixmaps required these values as parameters. Two methods were used to convert the values to the types needed by the SAIC code. The first was a simple type conversion as in the following example that converts a float number to an integer:

```
integer_number := integer( float_number );
```

The second method used unchecked conversion, a predefined generic function provided as part of the Ada language. This generic function had to be instantiated with a source type and a target type for each

Table 1: Parameter Conversion Rules

Outer Procedure Parameter Type	Mode	Inner Procedure Parameter Type	Method of Type Conversion
Widget	in	System.Address	XT.Widget_To_Addr(variable_name)[a]
	out		XT.Addr_To_Widget(variable_name)
AFS_Large_Natural[b] (integer ≥ 0)	in	AFS_Large_Natural	none
	out	System.Address	variable_name'address
String	in	System.Address	variable_name(1)'address
	out		
Boolean	in	AFS_Large_Natural	Boolean'pos(variable_name)
	out	System.Address	local_variable_name'address

[a]XT is an abbreviation of the Boeing package "X_TOOLKIT_INTRINSICS_OSF"
[b]This type is defined in the Boeing package "AFS_BASIC_TYPES"

conversion to be performed. An example instantiation to convert a variable of type "Display_Pointer" returned by Boeing's *Xt_Display* function to a variable of type "Display" for use in the SAIC routines follows:

```
function Display_Id_From_Xt_Display is new Unchecked_Conversion
                                  ( Source => XLIB.Display_Pointer,
                                    Target => X_Windows.Display );
```

The unchecked conversion utility allows a sequence of bits, an address in the above example, to be treated as a variable of two different types. However, this capability should be used with caution. As Cohen writes, "Abuse of this capability can subvert the elaborate consistency-checking mechanisms built into the Ada language and lead to improper internal representations for data"[2:804]. For the Saber user interface, however, this was the only way to pass certain variables created through the Boeing bindings as input parameters to the SAIC subroutines.

4.2.2 Problems With SAIC Data Structures.

Since the initial connection with the X server was made through the Xt Intrinsics via the Boeing bindings, and not through the SAIC code, several internal SAIC data structures were not initialized. Because these data structures were not initialized, some functions provided by the SAIC bindings could not be used.

Two of the functions that fell into this category were *Default_Depth* and *Root_Window*. The results returned by these functions were needed for the creation of the unit symbol pixmaps. To obtain these values, a binding was created for each function and added to the Boeing bindings. Before the values could be used by the SAIC subroutines, however, they had to be converted to the corresponding SAIC types. The value returned by *Root_Window* was converted using the unchecked conversion described in the previous section, while the value returned by *Default_Depth* was converted through simple type conversion.

5 Issues Affecting the Object-Oriented Design

We conducted a high-level design of the Saber user interface in the normal object-oriented fashion: identifying the primary objects and object classes, the object attributes, and the methods. However, peculiarities of the Xt Intrinsics require certain changes in the detailed design of the controlling modules. The objects and object classes by themselves are not useful until objects are instantiated. Objects can be instantiated

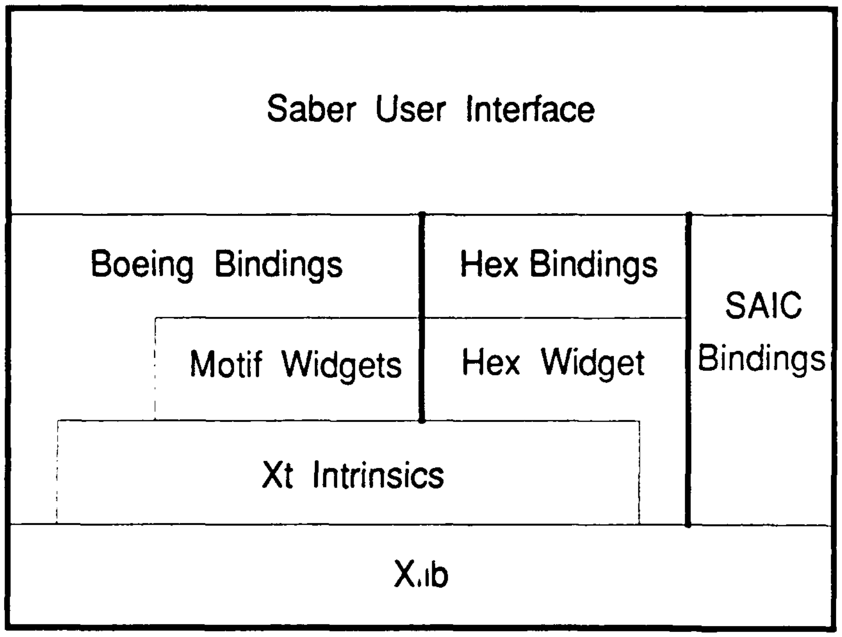

Figure 6: User Interface Relationship to the Ada Bindings

by another object or by some controlling module. In many programs, this controlling module is referred to as the main driver procedure. Unfortunately, this design does not work very well for programs which use the Xt Intrinsics. The reason for this is that the main procedure, after performing various initializations, typically enters a main loop through a call to the *XtMainLoop* function. This routine is an infinite loop that retrieves and dispatches events from the X event queue. When an event is dispatched for which a callback has been registered, processing in the main procedure is suspended and some other subroutine is executed. These callback procedures are the ideal place for object instantiation to take place.

The application programmer has a range of options available when developing the callback procedures. At one end of the spectrum, an individual callback can be written for every event that the program needs to be made aware of. For example, each button on a pulldown menu can have its own callback procedure. At the other extreme, a single callback can be written that handles all events. If this method is used the callback procedure must be able to determine what type of event triggered the callback. This can be accomplished by examining the event record created by the X Window System and passed as input to the callback procedure. Another way to determine what the event was is through the use of "client data" passed to the callback. This data is specified by the programmer when registering callbacks with the system. The client data can be of any type, and used for any purpose, that the programmer wishes. Thus, the client data could be used to identify why the callback procedure was entered.

When designing the arrangement of the callbacks, it is important to take into consideration the Motif widget hierarchy. The widgets used in an application program can be arranged in a hierarchy with all widgets, except for the top level widget, having exactly one parent widget. The widget id of the parent must be specified whenever a new widget is to be created. Thus, if a procedure is to create a new widget, it must have access to the widget's parent. One method of obtaining the parent's widget id is to receive it as an input parameter. By default, callback procedures always receive a parameter specifying the widget for which the callback was registered. If this widget is not the desired parent, then another alternative is to pass the parent's widget id as client data. Unfortunately, the parent's widget id may not be known at the time when the callback was registered. In this case, the only alternative is to make the parent widget globally available to the callback procedure. This, then, suggests that the callback procedures be grouped such that all needed parent widgets are visible. It makes sense to keep the group of global variables and associated callbacks as small as possible.

For the Saber user interface, several of the instantiated objects are either widgets themselves or require access to certain widgets. Since the selection of items on the menus presented to the user often involves the creation or manipulation of these objects, the design called for one or more controller packages that contained the callback procedures for each menu item. If the menu items can be separated into groups such that each group of associated callbacks deals with a single or small set of widgets and objects, then it is sometimes possible to develop a separate controller package for each group.

It is important to realize that the design is still object-oriented. The issue is when and where the objects are to be instantiated.

6 Limitations of the Bindings

The bindings written for Xlib, Xt Intrinsics and Motif widget set proved to be an indispensable part of the Saber user interface. While there were some weaknesses noted in the software, as a whole the bindings were able to directly or indirectly satisfy the requirements for the user interface. One problem common to the bindings is that they were designed for specific versions of the X software. Specifically, the SAIC bindings are for X11R3 and the Boeing bindings are for Motif V1.0.

6.1 Boeing Bindings

The first thing one notices when looking at the Boeing software is the lack of documentation. For the most part, the only documentation is in the form of section titles which separate the subroutines into topical categories. Thus, it would help if the application programmer is already familiar with the Xt Intrinsics and Motif widget set before trying to use the Boeing bindings. Furthermore, a few of the subroutines do not have nice, clean bindings to their corresponding C routines. These Ada subroutines use sparsely documented data structures that are defined within the bindings and that have no counterpart in the C code. It takes some time to learn what these data structures are for and how to use them properly.

A second weakness is that the bindings do not cover every Motif and Xt Intrinsics function. This fact is made clear in a "README" file that comes with the software. Some of the "missing" procedures can be added without too much difficulty. Other functions require a little more thought.

The third drawback to using the Boeing bindings is that they are currently tied to the Verdix Ada Development System (VADS) version 5.5 or higher. The bindings make use of the "C_Strings", "A_Strings", and "Command_Line" packages provided with the VADS library. The use of these packages restricts the portability of the application software. The "README" file included with the Boeing bindings indicates which modules would have to be changed to port the software to machines with different Ada compilers. However, the required changes should not be attempted by a novice Ada programmer.

6.1.1 Hardware Dependencies.

Even if a system does have VADS version 5.5 or higher, there is no guarantee that the Boeing bindings will work correctly. We found this out the hard way when attempting to use the bindings on a Sun 386i machine running VADS version 5.7 with Unix. Several test programs were written to gain familiarity with the bindings. However, they aborted with "Segmentation Faults" when executed. Analysis of the code showed that they were syntactically and semantically correct.

It was later determined that there were two problems, neither of which were caused by the Boeing bindings or the test programs. The causes of the problems were found in the August, 1991 edition of the *VADS Connection*. According to the Verdix Corporation, there are three potential problems areas to be aware of when writing programs that interface with C. These are parameter passing conventions, register usage, and parallelism. In this case, it was the first two areas that were causing the test program to abort.

The Verdix Corporation described the parameter passing conventions as follows[13:8]:

In many cases, C does not use the same parameter passing conventions as Ada. When calling C from Ada this is not a problem, because VADS automatically generates a C calling sequence whenever pragma INTERFACE is used. When calling Ada from C, however, there can be a problem. Verdix has implemented pragma EXTERNAL, which will cause an Ada subprogram to accept a C calling sequence, but this is only available in version 6.0.5 and above.

The problem encountered with register usage had to do with differences in the ways Ada and C use registers. According to the Verdix Corporation[13:8]:

For the 386...C expects the call to save and restore any registers it modifies, other than eax. Ada expects the caller to do the saving. This works fine when Ada calls C, but screws things up when C calls Ada. These register saves must be done manually, through the use of machine_code insertions.

At first glance, it did not appear that these issues would be causing the problems. It was obvious that Ada was making calls to C through the Boeing bindings, but it was not readily apparent that C was making any calls back to Ada. However, C was making calls to Ada inside of the *Xt_Main_Loop* procedure. Specifically, after the pushbutton is pressed, the C procedure *Xt_DispatchEvent* eventually causes control to be passed back to the Ada callback procedure that was registered with the pushbutton. It was at this point that the abovementioned problems caused the "Segmentation Fault".

However, we stress that this was not a problem with the Boeing bindings. Rather, it is inherent in the way callback procedures are dispatched. The test programs and the Boeing bindings worked correctly when the software was executed on a Sun Sparc Station 2.

6.2 SAIC Bindings

We also encountered a problem with the SAIC bindings, when we used them for the creation of the graphical symbols used to represent the air and land units in the Saber user interface. The problem was found when trying to read in the bitmap data created with the *Bitmap* editor provided with the X Window System software. This simple drawing program allows an application programmer to interactively create bitmap patterns. The pattern is saved in a special format that can be read in by an application program through calls to appropriate Xlib subroutines.

The *Bitmap* program outputs the bitmap data in groups of two hexadecimal digits. Thus, each of these two digit numbers is in the range 0 . . FF (or, in decimal, 0 . . 255). However, the SAIC bindings read each two digit number into an eight bit data structure called "Bit_Data" that can only handle numbers in the range $-2^7 . . 2^7 - 1$ (or, -128 . . 127). This means that any hexadecimal number greater than 7F is considered out of range.

Analysis of the errors revealed that the SAIC programmers made a previous attempt to correct this problem. We coded and tested a solution to the problem that solved the problem without creating any new errors.

7 Conclusion

In this paper we have presented a brief overview of the X Window System along with recent efforts for incorporating its use into Ada programs. One method involving the use of Ada bindings to X was presented in some detail. These bindings served as a example for developing new bindings for a user defined widget. While there were a few exceptions, most of the Ada subprograms bear a close resemblance to their C counterparts. Thus, anyone familiar with the calling sequences for the Xlib, the Xt Intrinsics and the Motif widget set should be able to understand the functionality of Ada programs that use the Boeing and SAIC bindings.

The impact of the X Window System on object-oriented design/programming was also discussed. While object definition is unaffected by X, new methods are needed for object instantiation and control. This is because of the main loop that is entered to obtain and dispatch events from the X server.

The continued use of the SAIC and Boeing bindings is encouraged for the development of graphical user interfaces in Ada.

Acknowledgements

The research for this paper was supported by a grant from the Air Force Wargaming Center, AU CADRE/WG, Maxwell AFB, AL, 36112.

References

[1] Ada Information Clearinghouse. *Available Ada Bindings*. Draft. Lanham, MD, October 1991.

[2] Cohen, Norman H. *Ada as a Second Language*. New York: McGraw-Hill, 1986.

[3] Hyland, Stephen J. and Mark A. Nelson. "Ada Bindings to the X Window System." Ada computer software source code, 1987.

[4] *Interface Standards Informal Technical Data, Ada Interfaces to X Window System*. Software Technology for Adaptable Reliable Systems (STARS) Contract F19628-88-D-0031, Publication No. GR-7670-1069(NP), Reston VA: Unisys Corporation, March 1989 (AD-A228820).

[5] Johnson, Eric F. and Kevin Reichard. *X Window Applications Programming*. Portland: MIS Press, 1989.

[6] Johnson, Eric F. and Kevin Reichard. *Power Programming ... Motif*. Portland: MIS Press, 1991.

[7] Jones, E. J. "Ada Bindings to the Xt Intrinsics and Motif Widget Set." Ada computer software source code, 1991.

[8] Klabunde, Capt Gary W. *An Animated Graphical Postprocessor for the Saber Wargame*. MS thesis. AFIT/GCS/ENG/91D-10, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1991.

[9] Myers, Brad A. and Mary Beth Rosson. "User Interface Programming Survey," *SIGCHI Bulletin*, *23*:27–30 (April 1991).

[10] Pountain, Dick. "The X Window System," *Byte*, *14*:353–360 (January 1989).

[11] Scheifler, Robert W. and Jim Gettys. "The X Window System," *ACM Transactions on Graphics*, *5*:79–109 (April 1986).

[12] Tevis, Jay-Evan J. II. *An Ada-Based Framework for an IDEF$_0$ CASE Tool Using the X Window System*. MS thesis, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1990 (AD-A189681).

[13] Verdix Corporation. *VADS Connection*. Technical Report. Chantilly, VA, August 1991.

[14] Wallnau, Kurt C. *Ada/Xt Architecture: Design Report*. Software Technology for Adaptable Reliable Systems (STARS) Contract F19628-88-D-0031, Publication No. GR-7670-1107(NP), Reston VA: Unisys Corporation, January 1990 (AD-A228827).

[15] Wallnau, Kurt C. and others. *Ada/Xt Toolkit, Version Description Document*. Software Technology for Adaptable Reliable Systems (STARS) Contract F19628-88-D-0031, Publication No. GR-7670-1133(NP), Reston VA: Unisys Corporation, July 1990 (AD-A229637).

[16] Young, Douglas. *The X Window System: Programming and Applications with Xt (OSF/Motif Edition)*. Englewood Cliffs NJ: Prentice Hall, 1990.

REPORT DOCUMENTATION PAGE	Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*

2. REPORT DATE
27 December 1991

3. REPORT TYPE AND DATES COVERED
Technical Report

4. TITLE AND SUBTITLE
Developing Object-Oriented User Interfaces in Ada with the X Window System

5. FUNDING NUMBERS

6. AUTHOR(S)
Gary W. Klabunde, Capt, USAF
Mark A. Roth, Maj, USAF

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology, WPAFB OH 45433-6583

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/EN-TR-91-8

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Air Force Wargaming Center
AU CADRE/WG
Maxwell AFB AL, 36112-5532

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Distribution Unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The graphical user interface has been accepted as being one of the most important parts of user interactive software projects. Until recently, however, the design and implementation of such interfaces in Ada was a long and difficult process. A solution to this problem was found with the introduction of the X Window System in the mid 1980's. These C libraries make it relatively easy to construct sophisticated user interfaces. In the last couple of years, several organizations have developed bindings to, or implementations of, the X Window System software for use in Ada programs. This paper discusses these efforts in general and concentrates on two sets of bindings that were used in the development of a graphical user interface for a computer wargame.

14. SUBJECT TERMS
Ada, X Windows, Motif, Ada Bindings

15. NUMBER OF PAGES
15

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

GENERAL INSTRUCTIONS FOR COMPLETING SF 298

The Report Documentation Page (RDP) is used in announcing and cataloging reports. It is important that this information be consistent with the rest of the report, particularly the cover and title page. Instructions for filling in each block of the form follow. It is important to ***stay within the lines*** to meet ***optical scanning requirements***.

Block 1. Agency Use Only *(Leave blank)*.

Block 2. Report Date. Full publication date including day, month, and year, if available (e.g. 1 Jan 88). Must cite at least the year.

Block 3. Type of Report and Dates Covered. State whether report is interim, final, etc. If applicable, enter inclusive report dates (e.g. 10 Jun 87 - 30 Jun 88).

Block 4. Title and Subtitle. A title is taken from the part of the report that provides the most meaningful and complete information. When a report is prepared in more than one volume, repeat the primary title, add volume number, and include subtitle for the specific volume. On classified documents enter the title classification in parentheses.

Block 5. Funding Numbers. To include contract and grant numbers; may include program element number(s), project number(s), task number(s), and work unit number(s). Use the following labels:

C -	Contract	**PR** -	Project
G -	Grant	**TA** -	Task
PE -	Program Element	**WU** -	Work Unit Accession No.

Block 6. Author(s). Name(s) of person(s) responsible for writing the report, performing the research, or credited with the content of the report. If editor or compiler, this should follow the name(s).

Block 7. Performing Organization Name(s) and Address(es). Self-explanatory.

Block 8. Performing Organization Report Number. Enter the unique alphanumeric report number(s) assigned by the organization performing the report.

Block 9. Sponsoring/Monitoring Agency Name(s) and Address(es). Self-explanatory.

Block 10. Sponsoring/Monitoring Agency Report Number. *(If known)*

Block 11. Supplementary Notes. Enter information not included elsewhere such as: Prepared in cooperation with...; Trans. of...; To be published in.... When a report is revised, include a statement whether the new report supersedes or supplements the older report.

Block 12a. Distribution/Availability Statement. Denotes public availability or limitations. Cite any availability to the public. Enter additional limitations or special markings in all capitals (e.g. NOFORN, REL, ITAR).

DOD -	See DoDD 5230.24, "Distribution Statements on Technical Documents."
DOE -	See authorities.
NASA -	See Handbook NHB 2200.2.
NTIS -	Leave blank.

Block 12b. Distribution Code.

DOD -	Leave blank.
DOE -	Enter DOE distribution categories from the Standard Distribution for Unclassified Scientific and Technical Reports.
NASA -	Leave blank.
NTIS -	Leave blank.

Block 13. Abstract. Include a brief *(Maximum 200 words)* factual summary of the most significant information contained in the report.

Block 14. Subject Terms. Keywords or phrases identifying major subjects in the report.

Block 15. Number of Pages. Enter the total number of pages.

Block 16. Price Code. Enter appropriate price code *(NTIS only)*.

Blocks 17. - 19. Security Classifications. Self-explanatory. Enter U.S. Security Classification in accordance with U.S. Security Regulations (i.e., UNCLASSIFIED). If form contains classified information, stamp classification on the top and bottom of the page.

Block 20. Limitation of Abstract. This block must be completed to assign a limitation to the abstract. Enter either UL (unlimited) or SAR (same as report). An entry in this block is necessary if the abstract is to be limited. If blank, the abstract is assumed to be unlimited.